

# NAVAL POSTGRADUATE SCHOOL

## Monterey, California



## THESIS

### DESIGN AND IMPLEMENTATION OF REAL-TIME MONITOR FOR THE OBJECT-ORIENTED INTERFACE

by

Erhan Senocak

December 1995

Thesis Advisor:  
Thesis Co-Advisor

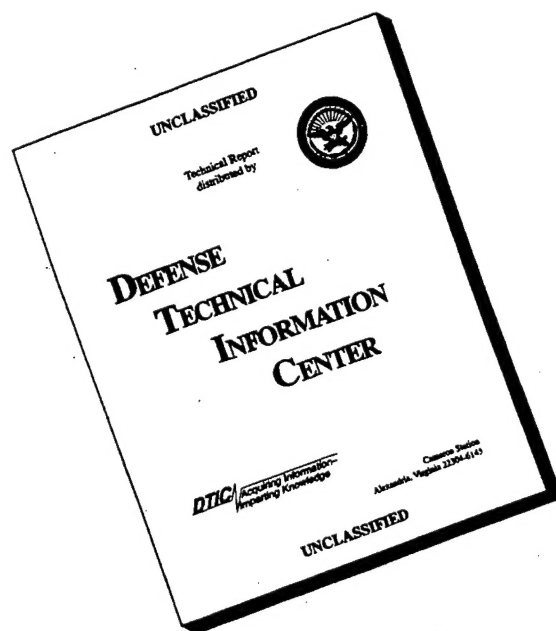
David K. Hsiao  
C. Thomas Wu

Approved for public release; distribution is unlimited.

19960401 012

DTIC QUALITY INSPECTED 1

# DISCLAIMER NOTICE



**THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.**

**REPORT DOCUMENTATION PAGE**Form Approved  
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time reviewing instructions, searching existing data sources gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

**1. AGENCY USE ONLY (Leave Blank)****2. REPORT DATE**  
December 1995**3. REPORT TYPE AND DATES COVERED**  
Master's Thesis**4. TITLE AND SUBTITLE**

Design and Implementation of Real-Time Monitor for the Object-Oriented Interface

**5. FUNDING NUMBERS****6. AUTHOR(S)**

Erhan Senocak

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**Naval Postgraduate School  
Monterey, CA 93943-5000**8. PERFORMING ORGANIZATION  
REPORT NUMBER****9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)****10. SPONSORING/ MONITORING  
AGENCY REPORT NUMBER****11. SUPPLEMENTARY NOTES**

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

**12a. DISTRIBUTION / AVAILABILITY STATEMENT**

Approved for public release; distribution is unlimited.

**12b. DISTRIBUTION CODE****13. ABSTRACT (Maximum 200 words)**

In a stand-alone database management system (DBMS), one of the key components is the real-time monitor (RTM) which handles database accesses and responses at run time. In the Multimodel, Multilingual and Multibackend Database Management System (M<sup>3</sup>DBMS) developed at the Laboratory for Database System Research in the Naval Postgraduate School, there is also the need of a RTM in order to link a specific Data Model and Data Language Interface to the Kernel DBMS. The problem addressed by this thesis is to design and implement a RTM for the Object-Oriented Interface in M<sup>3</sup>DBMS.

In this interface each object-oriented (OO) query is converted into the equivalent Attribute-Based Data Language (ABDL) queries. However, due to the complexity of the OO operations there is no way to produce these ABDL queries in complete and executable forms. Much of the information needed for the completion and execution of the ABDL queries is provided by the previous ABDL queries. The approach was to develop a RTM which oversees the execution of previous ABDL queries, receives the intermediate results from these queries, and completes the subsequent ABDL queries for execution in the Kernel.

The result of this thesis is a RTM which executes the OO query as directed by the compiler of object-oriented data manipulation language (OODML). Once the OO query is parsed by the OODML compiler, it is transformed into the equivalent ABDL queries and a series of pseudocode in compliance with the protocol between the OODML compiler and the RTM. The RTM executes the operations specified by the pseudocode by using its built-in functions. However, for the execution of the ABDL queries, it communicates with the Kernel DBMS.

**14. SUBJECT TERMS**Real-Time Monitor  
Object-Oriented Data Manipulation Language  
Kernel Database Management System**15. NUMBER OF PAGES**

65

**16. PRICE CODE****17. SECURITY CLASSIFICATION  
OF REPORT**

Unclassified

**18. SECURITY CLASSIFICATION  
OF THIS PAGE**

Unclassified

**19. SECURITY CLASSIFICATION  
OF ABSTRACT**

Unclassified

**20. LIMITATION OF ABSTRACT**

UL



Approved for public release; distribution is unlimited

**DESIGN AND IMPLEMENTATION OF REAL-TIME MONITOR  
FOR THE OBJECT-ORIENTED INTERFACE**

Erhan Senocak  
Lieutenant Junior Grade, Turkish Navy  
B.S. Turkish Naval Academy, 1989

Submitted in partial fulfillment of the  
requirements for the degree of

**MASTER OF SCIENCE IN COMPUTER SCIENCE**

from the

**NAVAL POSTGRADUATE SCHOOL**

**December 1995**

Author:



---

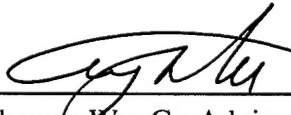
Erhan Senocak

Approved by:



---

David K. Hsiao, Thesis Advisor



---

C. Thomas Wu, Co-Advisor



---

Ted Lewis, Chairman  
Department of Computer Science



## ABSTRACT

In a stand-alone database management system (DBMS), one of the key components is the real-time monitor (RTM) which handles database accesses and responses at run time. In the Multimodel, Multilingual and Multibackend Database Management System (M<sup>3</sup>DBMS) developed at the Laboratory for Database System Research in the Naval Postgraduate School, there is also the need of a RTM in order to link a specific Data Model and Data Language Interface to the Kernel DBMS. The problem addressed by this thesis is to design and implement a RTM for the Object-Oriented Interface in M<sup>3</sup>DBMS.

In this interface each object-oriented (OO) query is converted into the equivalent Attribute-Based Data Language (ABDL) queries. However, due to the complexity of the OO operations there is no way to produce these ABDL queries in complete and executable forms. Much of the information needed for the completion and execution of the ABDL queries is provided by the previous ABDL queries. The approach was to develop a RTM which oversees the execution of previous ABDL queries, receives the intermediate results from these queries, and completes the subsequent ABDL queries for execution in the Kernel.

The result of this thesis is a RTM which executes the OO query as directed by the compiler of object-oriented data manipulation language (OODML). Once the OO query is parsed by the OODML compiler, it is transformed into the equivalent ABDL queries and a series of pseudocode in compliance with the protocol between the OODML compiler and the RTM. The RTM executes the operations specified by the pseudocode by using its built-in functions. However, for the execution of the ABDL queries, it communicates with the Kernel DBMS.





## TABLE OF CONTENTS

I.	INTRODUCTION .....	1
II.	THE NEED FOR THE REAL-TIME MONITOR IN THE OBJECT-ORIENTED INTERFACE.....	5
III.	THE INTERACTION WITH OTHER SOFTWARE MODULES .....	9
	A. THE INTERACTION WITH THE OODML COMPILER.....	9
	B. THE INTERACTION WITH THE KERNEL DATABASE SYSTEM.....	12
	C. INTERACTION WITH THE KERNEL FORMATTING SYSTEM.....	13
IV.	DESIGN AND IMPLEMENTATION ISSUES .....	15
	A. THE OVERALL DESIGN OF THE REAL TIME MONITOR.....	15
	B. THE IMPLEMENTATION OF THE SECONDARY OPERATIONS.....	17
	C. THE IMPLEMENTATION OF THE PRIMARY OPERATIONS.....	20
V.	CONCLUSIONS.....	25
	A. LIMITATIONS .....	25
	B. FUTURE RESEARCH .....	25
	APPENDIX A- THE DEFINITIONS FOR THE PSEUDOCODE.....	27
	APPENDIX B- THE SAMPLE TRACE OF A QUERY .....	29
	APPENDIX C- SOURCE CODE FOR THE RTM.....	31
	LIST OF REFERENCES.....	51
	INITIAL DISTRIBUTION LIST.....	53



## LIST OF FIGURES

Figure 1. Interfaces, Databases and Schemas in M3DBMS with CMAC .....	2
Figure 2. The Role of the RTM between the Interface and the Kernel .....	7
Figure 3. The Data Flow between the RTM and the Other Softwares .....	10
Figure 4. The Flowchart for rtm_exec() .....	16
Figure 5. The Flowchart for the query_exec() .....	21

## I. INTRODUCTION

Different Database Management Systems (DBMS) are derived from different data models and data languages. These DBMS create heterogeneous databases. To increase the effective utilization of all these data stored in various databases and to prevent the data duplication of one database into the other database, the researcher begins to search a way for the interoperability of heterogeneous databases and their DBMS.

Although it is still in experimental phase, the interoperable solution developed at the Laboratory for Database System Research in the Naval Postgraduate School may turn out to be the most promising one, since it removes the limitations of the other solutions offered so far. For a detailed explanation of the other solutions and their limitations, the reader is to refer to [Ref. 1]. Our interoperable solution is the Multimodel, Multilingual and Multibackend Database System ( $M^3$ DBMS) with the Cross-Model-Accessing Capabilities (CMAC), where all the heterogeneous databases are integrated into a kernel database. This integration is done automatically by the  $M^3$ DBMS with the CMAC and is transparent to the user. More specifically, the system enables the user to create a database in the data model preferred, to access and manipulate not only the user's own database but also the other users' databases. For example, based on the other data models, the relational user of a relational database may access a hierarchial database as if it is a relational database. The *multimodel*, *multilingual* and *CMAC* characteristics ensure the interoperability of heterogeneous databases; the *multibackend* characteristic ensures the parallel operations and huge storage of the databases. In this system there is an interface for each distinct pair of data model and data language. For the database creation, the interface converts a database created in a specific data model into the kernel (i.e., attribute-based) data model format and creates for it the kernel database organized as attribute-value pairs. For the manipulation of the database, the interface also converts the queries in the user's favorite data language into the kernel (i.e., attribute-based) data language (ABDL). The entire system is depicted in Figure 1.

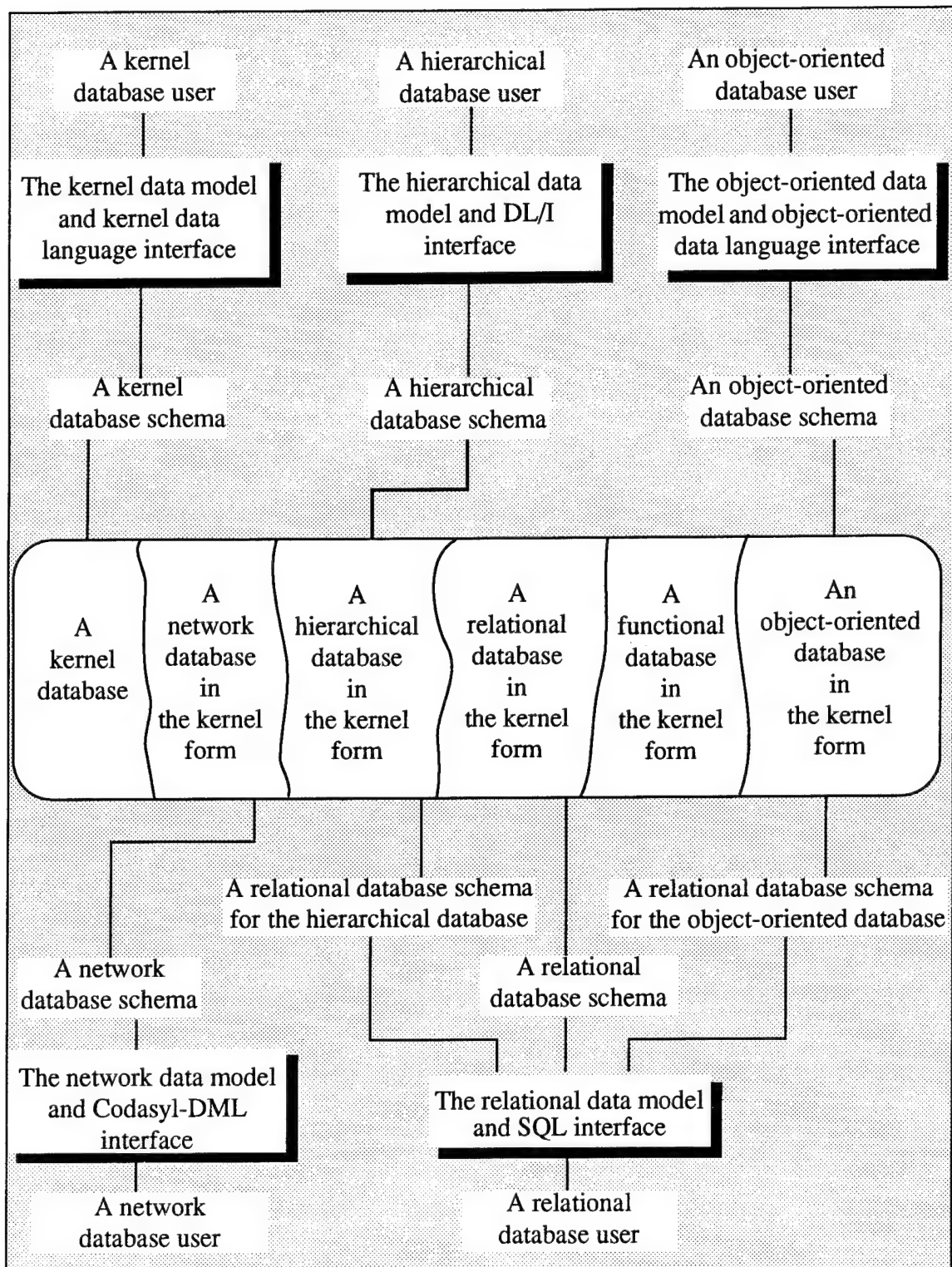


Figure 1: Interfaces, Databases and Schemas in M³DBMS with CMAC

Beginning in the early 90's the Object-Oriented Database Management Systems (OODBMS) became one of the major trends in the database world. Many "next-generation" applications, such as computer-aided design and manufacturing systems, computer-aided software engineering, multimedia and hypermedia information systems, and artificial intelligence expert systems require databases that can support objects of a variety of types with ability to express complex relationships among objects [Ref. 2]. Since conventional database systems (network, hierarchial and relational models) are often inadequate to support the requirements of these applications, OODBMS have gained more and more popularity. Although some vendors are enriching RDBMS and claiming that RDBMS are as powerful as OODBMS, there is no doubt that the latter with rich and complex semantic database constructs are superior to all the previous DBMS.

Before the present thesis work, M<sup>3</sup>DBMS has supported attribute-based, hierarchial, network, relational, and functional data models and their corresponding data languages. With the completion of this and other theses, the newly emerging data model and data language, namely, the object-oriented data model and the object-oriented data language, are incorporated as an object-oriented data model/language interface into the M<sup>3</sup>DBMS.

The objective of this thesis research is the design and implementation of a real-time monitor (RTM) for object-oriented database management. It supervises the execution of well-formed ABDL queries being produced by the compiler of the object-oriented data manipulation language (OODML). It also completes other nearly-executable ABDL queries for execution. Finally, it presents the answer of the query to the kernel formatting system (KFS). Detailed explanations of the well-formed and nearly-executable ABDL queries are given in the following chapters. More specifically, in the remaining chapters of this thesis, the need for the RTM in the Object-Oriented Interface is articulated in Chapter II. In Chapter III, the interactions of the RTM with the other modules of the interface and the kernel database are expounded. In Chapter IV, the design and the implementation of the

RTM are detailed. The conclusion of the thesis is included in Chapter V. Following the last chapter, there are appendices on the pseudocode definitions, the sample trace of a query and the source code of the RTM.

## II. THE NEED FOR THE REAL-TIME MONITOR IN THE OBJECT-ORIENTED INTERFACE

During the design phase of the object-oriented interface, one of the important issues is how to convert the object-oriented operations of the interface into the equivalent ABDL functions in the kernel. For a detailed explanation of the object-oriented operations see [Ref. 3]. Although simple ABDL functions, namely INSERT, DELETE, UPDATE, RETRIEVE and RETRIEVE COMMON are capable of supporting complex object-oriented operations, there is no way to produce these ABDL functions in complete and executable forms. Much of the information needed for the completion and execution of ABDL functions must be provided by previous ABDL functions and computations. Thus, there is the need of a real-time monitor, which oversees the execution of the previous ABDL functions; receives the intermediate results from these functions, and completes the subsequent ABDL functions for their execution in the kernel.

Another option might be to modify the kernel in order to handle the situation defined above. Since the processes in the kernel are being shared by the other interfaces of M<sup>3</sup>DBMS and the code regarding the implementation of the kernel database management system is very long and complex, any modification of the kernel would be very time consuming and may result in unexpected problems in building up a robust kernel database system. The reader is to refer to [Ref. 4] and [Ref. 5] for a detailed explanation of the kernel database system. So, due to the risk of introducing errors into the kernel database system and complicating the implementation of the object-oriented interface, the use of a layer of software between the object-oriented interface and the kernel is a much easier and rational solution.

Besides, in all the other interfaces of M<sup>3</sup>DBMS, the need of a real-time monitor (called as the kernel controller) is inevitable. In general, the kernel controller submits the equivalent ABDL queries to the kernel and receives the answers from the kernel. So, the



kernel controller acts like a real-time monitor of a stand-alone DBMS, handling the database accesses at the run time.

Architectually, the RTM in the object-oriented interface is similar to the kernel controllers in the other interfaces. However, the RTM in the object-oriented interface facilitates object-oriented operations, completes all the incomplete assembled ABDL operations, and interacts with the kernel database system on behalf of the object-oriented interface.

The role of the RTM between the object-oriented interface and the kernel is depicted in Figure 2.

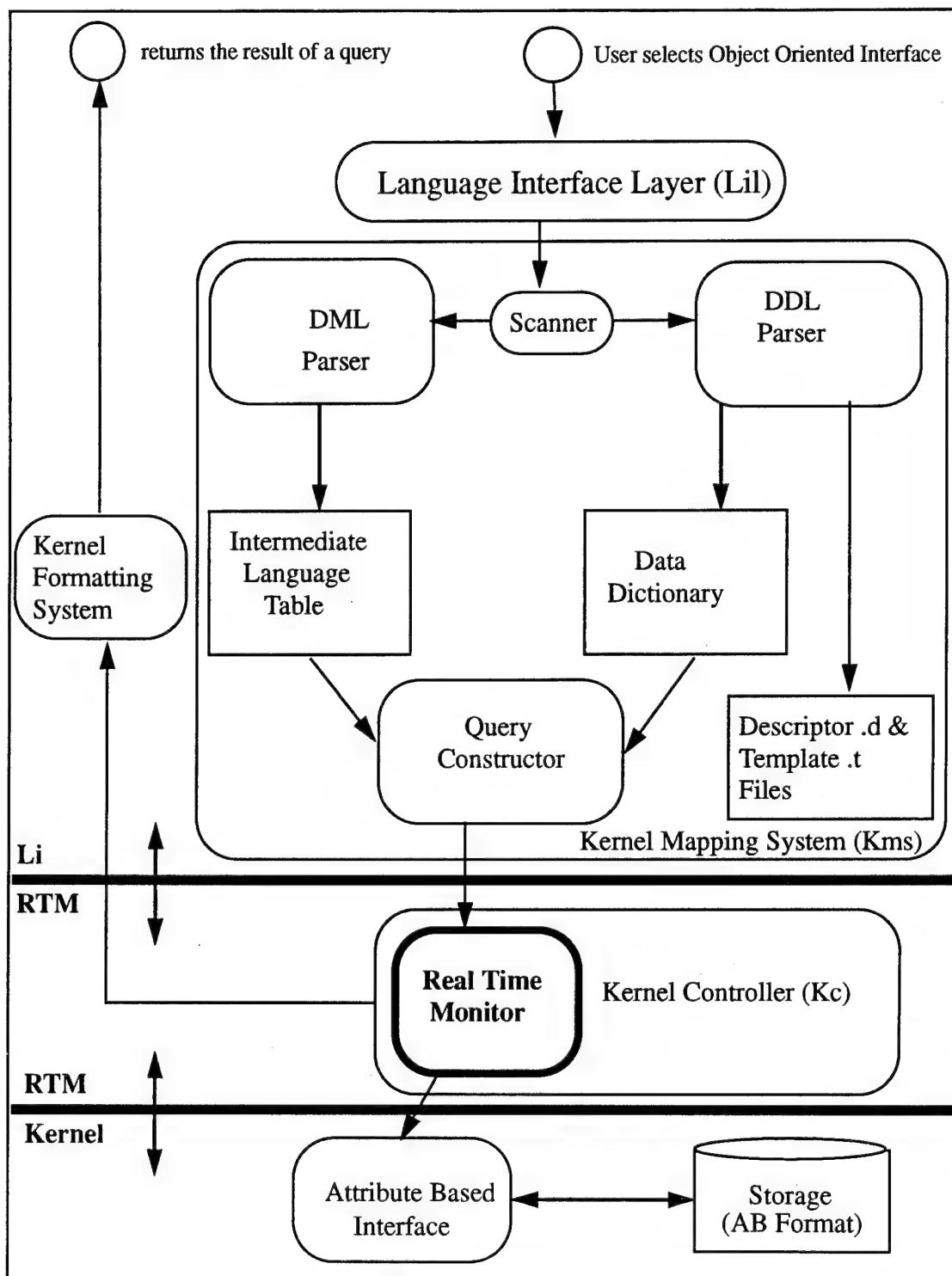


Figure 2. The Role of the RTM between the Interface and the Kernel



### III. THE INTERACTION WITH OTHER SOFTWARE MODULES

Before describing the design and the implementation of the RTM, depicting the relationship of the RTM with the other modules of the interface and the kernel, that is, defining the input and the output of the RTM will make the architecture of the RTM much more readable and understandable.

Basically, the RTM communicates with three software modules. The first is the *query constructor*, the last component of the OODML compiler, which is responsible for preparing the input file to the RTM, namely *query\_f*. The RTM executes the *query\_f* line by line and communicates with the kernel for the execution of each ABDL query in this file. So, the *kernel* is the second software module that the RTM interacts with. Once the execution of the whole query file is completed, the RTM checks if it has created an output file that has to be displayed to the user. If it has, then it activates the *kernel formatting system*. It is the third software module to which the RTM is related. The data flow between the RTM and the other software modules is depicted in Figure 3.

In the following sections, the interaction of the RTM with the other software modules is expounded.

#### A. THE INTERACTION WITH THE OODML COMPILER

The input file that the RTM processes on, is prepared by the OODML compiler. This file consists of the translation of an object-oriented query into the attribute-based queries. The translation process in the OODML compiler takes several steps. First, each object-oriented query is tokenized using the tool LEX. Once all the tokens have been identified, the tool YACC is used as a parser in order to check the syntactic and semantic correctness of the query. The parser also constructs the intermediate-language table and the symbol table. Finally, the query constructor produces the equivalent ABDL queries accessing the intermediate-language table, the data dictionary, and the symbol table [Ref.7]. Some of these ABDL queries are *well-formed* and some of them are *nearly-executable*. The query constructor also produces the pseudocode with respect to the type of

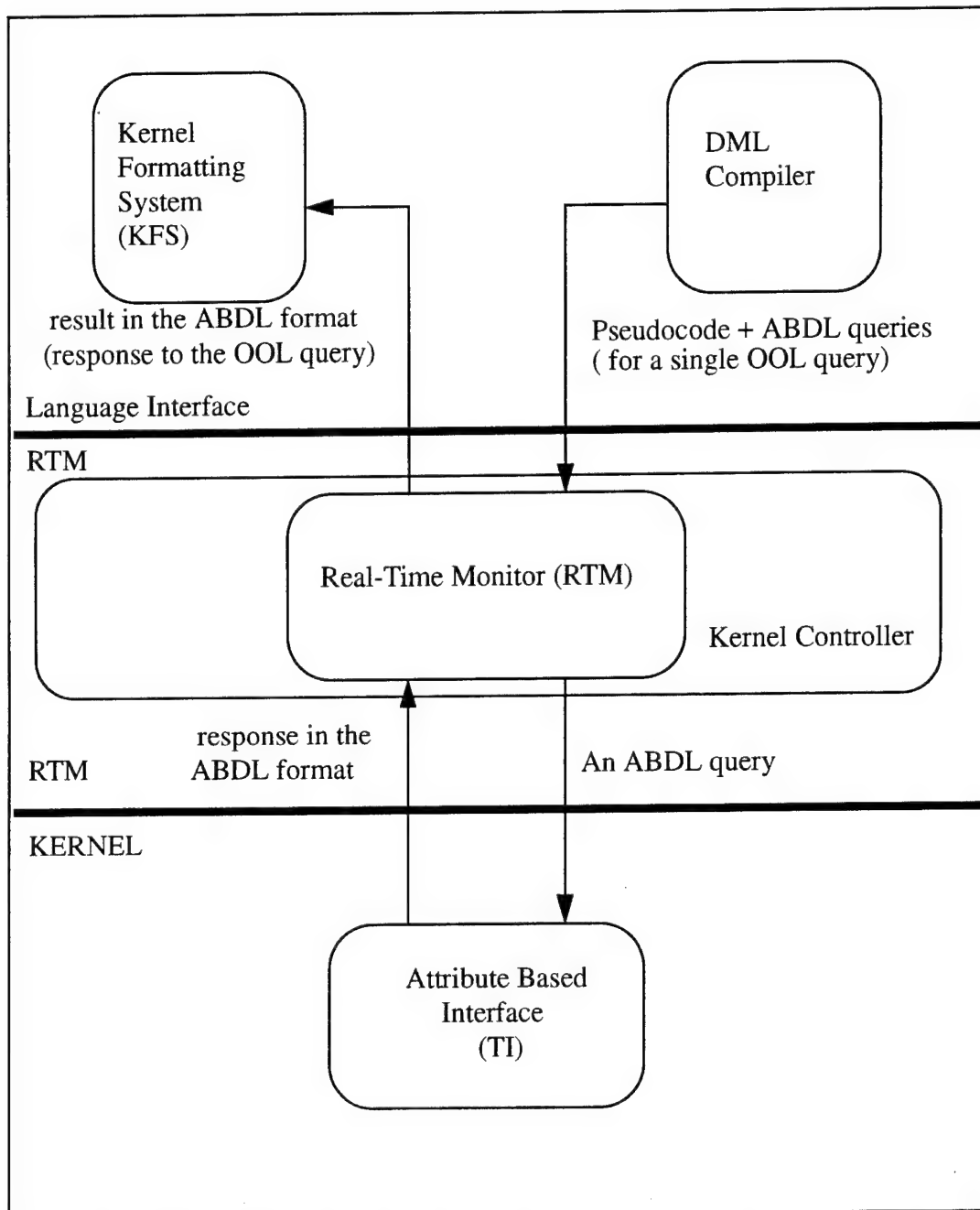


Figure 3: The Data Flow between the RTM and the Other Softwares

the operations in the object-oriented query. Finally, all these statements, i.e., well-formed and nearly-executable ABDL queries and the pseudocode are packed and sent to the RTM as a text file *query\_f*.

As stated above, the *query\_f* is composed of three types of statement; well-formed ABDL queries, nearly-executable ABDL queries, and the pseudocode. A well-formed ABDL query is the ABDL query which is complete and ready to be sent to the kernel database for execution. A nearly-executable ABDL query is an ABDL query which is complete in terms of the syntax of the ABDL but, incomplete in terms of the values of its arguments. In other words, the ABDL query includes a variable that has to be substituted with its value or values before execution. The value of the variable is determined from the execution of the previous ABDL queries. The presence of the nearly-executable ABDL queries is due to the fact that the compiler can not build all the ABDL queries in well-formed way. Since some ABDL queries are dependent on the results of the previous ABDL queries, there is no way for the compiler to know the answer of an ABDL query in advance in order to prepare the next query with the result of the previous one. Because the RTM is present in the course of the execution of the queries, the completion of the nearly-executable queries for their execution becomes a natural task of the RTM.

The RTM always works on the object identifier (OID) values received from the kernel database in response to the ABDL queries. Although the existence of the OIDs is hidden from the user, it is the only way to keep track of objects, since for each object there is a unique OID assigned by the RTM. Consequently, all the computations and the manipulations performed by the RTM make use of the OID values.

As outlined in [Ref. 3], the object-oriented operations are much more complex and may involve more primary ABDL operations. The *for-loop*, *assignment*, *display* are some of the object-oriented operations that do not have equivalent ABDL functions, since they are not storage and retrieval operations. So, to implement these operations and to complete nearly-executable ABDL queries for execution, there is a protocol between the RTM and the compiler, namely, pseudocodes. Each pseudocode is a unique single-character used for

a particular operation. They are the only means to activate the corresponding functions in the RTM so that some of the object-oriented operations are performed without the kernel. The reader is to refer to Appendix A for the pseudocode definitions.

Since the RTM can be assumed as a bridge between the object-oriented language interface and the kernel, the way the communication is handled is very important. The next section will focus on this issue.

## **B. THE INTERACTION WITH THE KERNEL DATABASE SYSTEM**

One of the responsibilities of the RTM is to handle the inter-communication between the kernel database system and the object-oriented interface. The communication with the kernel is crucial because it is the only possible way to send an ABDL query to the kernel and get the result from the kernel. All communications are accomplished via the Test Interface (TI) process which is one of the six processes residing in the kernel.

There are two main TI\_ functions that the RTM utilizes for ABDL query execution. All other TI\_ functions are called by way of these two functions. Once the RTM determines that an ABDL query is ready for execution, it calls the function *TI\_S\$TrafUnit(dbid, trafunit)* in the following form:

*TI\_S\$TrafUnit(oool\_ptr -> oi\_curr\_db.cdi\_dbname, request)*

The structure *oool\_ptr -> oi\_curr\_db.cdi\_dbname* holds the database identifier (dbid) and the variable *request* holds the traffic unit (trafunit) which is the ABDL query itself. The *dbid* determines the database that the kernel is going to access.

Following this function call the RTM calls the function *TI\_RTM\_chk\_res\_left()* to ensure that the request has been processed and the results from the kernel system have been received. The function *TI\_RTM\_chk\_res\_left()* communicates with the rest of the kernel system and receives the message about the condition of the request. If errors exist, the function *TI\_R\$ErrorMessage()* is called to get the error message. However, if no error occurs, the function *TI\_R\$ReqRes()* is called to receive the response from the other processes of the kernel system. The response buffer is then checked to see if this is the last

response. If it is the last response, the results are then loaded into the file *response\_f* through the calls to the functions *TI\_ReqRes\_RTMoutput()* and *TI\_print\_RTMReqRes()*, respectively [Ref. 5].

Depending on the type of the primary operation, the process on the *response\_f* is handled accordingly by the RTM. If the primary operation is a RETRIEVE then the file *response\_f* is opened. All the responses are OID values. They are used by the secondary and/or primary operations followed. But if the primary operation is a RETRIEVE where the result is to be displayed to the user, namely, DISPLAY, then the content of the *response\_f* is stored in the file *output\_f*. In this case, the content of the *response\_f* consists of attribute-value pairs, not OID values. The responses to all the other DISPLAY operations are also appended to the *output\_f*. When the execution of the *query\_f* is completed, the RTM sends this file to the KFS.

### C. INTERACTION WITH THE KERNEL FORMATTING SYSTEM

Once the execution of the *query\_f* is completed, the RTM checks if it has created the *output\_f*. As mentioned in the previous section, this file is created in case the user wants the data retrieved from the database to be displayed on the screen. Since the data retrieved from the database is in the attribute-based format, the KFS reformats them in a tabular format and displays to the user in an object-oriented fashion. Since there might be more than one DISPLAY operations in an object-oriented query, the responses to each DISPLAY operation are separated by an asterisk (\*) in the *output\_f* by the RTM. On the basis of the separators, the KFS can determine whether it should to create a new table with new attribute names as the column headings. For a detailed explanation regarding the implementation of the KFS, refer to [Ref. 4].





## IV. DESIGN AND IMPLEMENTATION ISSUES

### A. THE OVERALL DESIGN OF THE REAL TIME MONITOR

Once the function *real\_time\_monitor()* is activated with the *query\_f*, the RTM takes the control in the object-oriented interface and begins to execute the *query\_f* line by line. There are two types of operations that the RTM handles: Primary operations and secondary operations. Primary operations are related to the execution of the ABDL functions. The operations regarding data retrieval and manipulation, i.e., the operations that the RTM has to communicate with the kernel are referred to as *primary operations*. The operations internal to the RTM for the execution of the pseudocode are called as *secondary operations*.

As stated before, all types of pseudocode are symbolized by one character and placed at the beginning of each line in the *query\_f*. So, the first character of each line determines the action to be taken by the RTM. After the *real\_time\_monitor()* is called, the program flow shifts to the main controller function *rtm\_exec()*. The body of this function is a *switch* statement. Since the lines are read character by character, once the first character of a line is read, it is compared with one of the characters defined in the *switch* statement and depending on whether it is a pseudocode or an ABDL query, the corresponding secondary operation or the primary operation is executed. There is a function for each type of pseudocode. The flowchart for the function *rtm\_exec()* is demonstrated in Figure 4. As seen in Figure 4, the *rtm\_exec()* is responsible for calling the functions regarding secondary operations. In case the first character of a line is a left bracket ([), the *rtm\_exec()* calls the function *query\_exec()* that is responsible for the execution of the primary operations.

The relationship between the secondary and primary operations are handled through the flags. Depending on the type of the pseudocode there might be a warning such as *substitution*, *double substitution*, *assignment* for the ABDL query in the next line. In case of these warnings, the corresponding flags are set, so that in the next line, the status of these flags is checked first and the necessary action is taken before executing the ABDL query. At the end of the execution of the ABDL query the corresponding flags are reset.

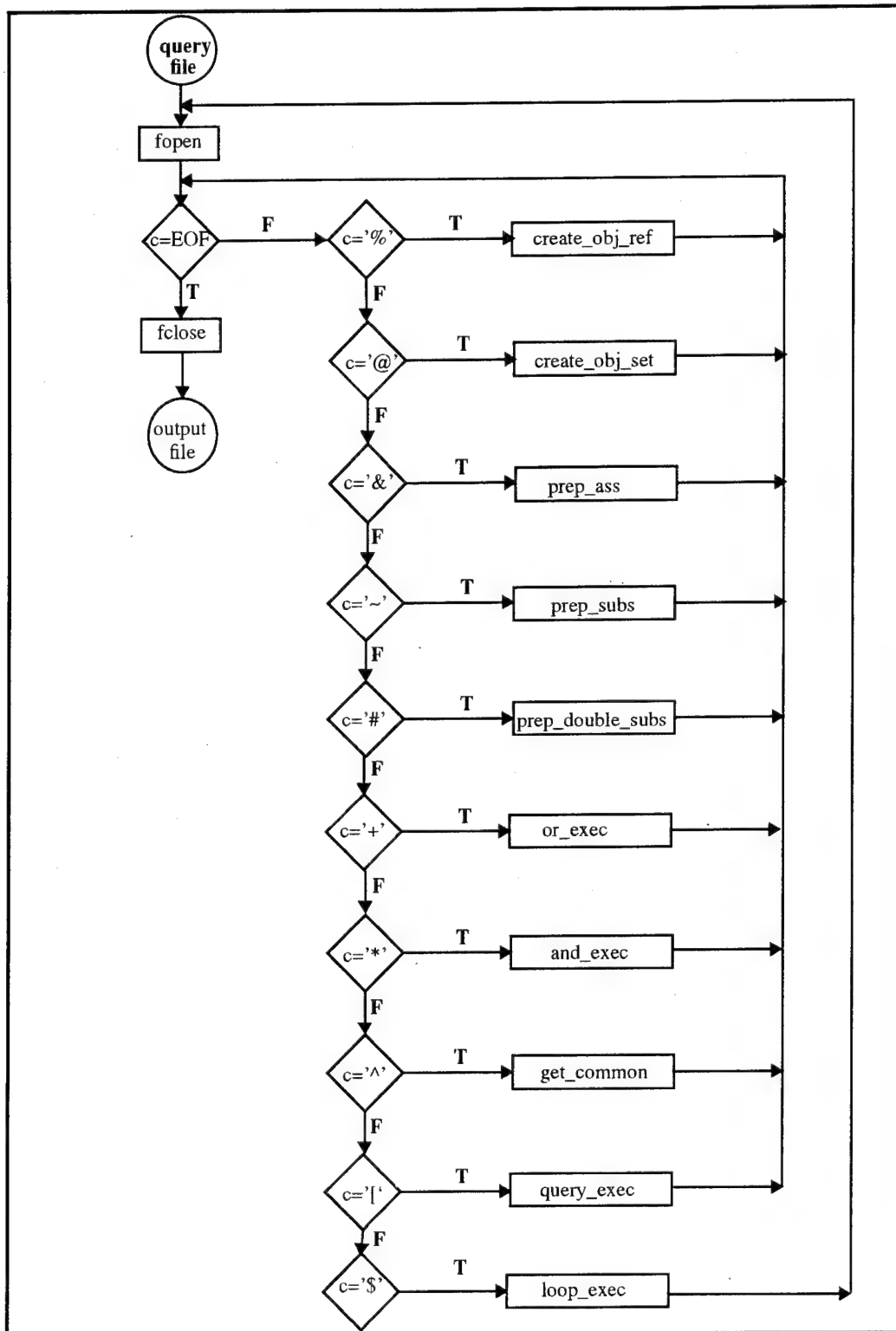


Figure 4: The Flowchart for rtm\_exec()

The reader is to refer to Appendix B for the sample trace of a query. The script file in Appendix B consists of the original object-oriented query, the *query\_f* produced by the query constructor, the steps taken by the RTM during the execution of the *query\_f*, and the end-result displayed by the KFS.

The software for the RTM is implemented in C programming language. The *rtm\_def.h* and the *rtm.c* reside in the directory *db11/u/mdbs/greg/CNTRL/TI/LangIF/src/Obj/Kc*. The reader is to refer to Appendix C for the source code.

## B. THE IMPLEMENTATION OF THE SECONDARY OPERATIONS

As seen in Figure 4, each secondary operation is implemented as a function in the RTM. In the following lines, each one of these functions is described.

**create\_obj\_ref():** This function is used to create object reference (*obj\_ref*) and store it in the data structure defined below. The name and the number of the *obj\_refs* are defined in the *query\_f*. The data structures used for this function are as follows:

```
struct obj_ref
{
    char name[3];
    char data[20];
};
struct obj_ref list_obj_ref[30];
```

The field *name* of the struct *obj\_ref* is used to store the name of the *obj\_ref* and the field *data* is used to store the OID value. Since there may be more than one *obj\_refs* in the *query\_f*, they are stored in the array *list\_obj\_ref[30]*.

**create\_obj\_set():** This function has the same functionality as the *create\_obj\_ref()* does. The only difference is that the *data* field of the struct *obj\_set* is defined as an array of array of characters since an object set (*obj\_set*) may have more than one OID values. The data structures defined for this function are as follows:

```
struct obj_set
```

```

{
    char name[3];
    char data[30][20];
};

struct obj_set list_obj_set[30];

```

**prep\_ass():** This function deals with the intermediate steps necessary for the assignment. First, it reads the name of the variable that the assignment will be done. This is either an obj\_ref or an obj\_set. Next, the index of this variable in the corresponding list is found and stored in the global variable *index*. Depending on whether it is an obj\_ref or an obj\_set, the flag *o\_r\_ass* or *o\_s\_ass* is set, correspondingly. The assignment is required by the primary operation RETRIEVE, and the secondary operations AND, OR, and GET\_COMMON. It is used for assigning the OID values to the *data* field of the obj\_ref/obj\_set referenced by the *index*.

**prep\_subs():** This function handles the intermediate steps necessary for any substitution in an ABDL query. First, it reads the name of the variable to be substituted with the OID value in its *data* field. The substitution variable may be either an obj\_ref or an obj\_set. Next, the index of this variable in the corresponding list is found and with respect to the type of the variable the flag *flag\_subs\_or* or the flag *flag\_subs\_os* is set, correspondingly. The data structures defined for this function are as follows:

```

struct obj_ref_set_info
{
    char name[3];
    int index;
};

struct obj_ref_set_info subs_obj_var;

```

**prep\_double\_subs():** This function handles the intermediate steps necessary for double substitution in an ABDL query. The difference between this function and the prep\_subs() is that this one allows for two substitutions in an ABDL query but the type of

the substitution variables have to be *obj\_ref*. The function reads the name of the variables from the *query\_f*. After locating their index in the *list\_obj\_ref*, it sets the flag *flag\_double\_subst*. The data structure defined for this function is as follows:

```
struct obj_ref_set_info list_subs_obj_ref[2];
```

**or\_exec():** This function computes the union of the OID values in two distinct *obj\_sets* and stores them in a new *obj\_ref/obj\_set*. After locating the index of each *obj\_set* subject to the union, the function determines the type of the assignment variable depending on whether the flag *o\_r\_ass* or *o\_s\_ass* is set. If the type of the assignment variable is an *obj\_ref*, then the first OID value of the first *obj\_set* is assigned to the data field of the assignment variable and, the function terminates. If the type of the assignment variable is an *obj\_set*, first, all the OID values in the first *obj\_set* are assigned. Then, each OID value in the second *obj\_set* is compared to the OID values in the first *obj\_set*. If it does not match with any of them then it is added to the data field of the assignment variable. The data structure used for this function is as follows:

```
struct obj_ref_set_info list_or_var[2];
```

**and\_exec():** This function computes the conjunction of the OID values in two distinct *obj\_sets* and stores them in a new *obj\_ref/obj\_set*. The steps performed at the beginning are same as in the function *or\_exec()*. After the function begins to compare the OID values of the first *obj\_set* to the second *obj\_set*'s, the action taken depends on the type of the assignment variable. If it is an *obj\_ref*, the first OID value that matches is stored in the assignment variable and, the function terminates. If the type of the assignment variable is an *obj\_set*, the comparison between the *obj\_sets* continue and all the OID values that are equal, are assigned in the specific *obj\_set*. The data structure defined for this function is as follows:

```
struct obj_ref_set_info list_and_var[2];
```

**get\_common():** This function determines the OID values in an *obj\_set*, that are repeated for at least the number of OID values in another *obj\_set* or in an *obj\_ref*, and it assigns these OID values in a new *obj\_ref/obj\_set*. The first variable the function reads is

always an *obj\_set*, and the second one may be either an *obj\_set* or an *obj\_ref*. After their index is determined, the type of the assignment variable is checked. If it is an *obj\_set* and the second variable is an *obj\_ref*, then all the OID values in the first variable are assigned to the *data* field of the assignment variable. If the second variable is an *obj\_set*, then the OID values in the first *obj\_set*, that are repeated for at least the number of OID values in the second *obj\_set*, are assigned to the *data* field of the assignment variable. If the type of the assignment variable is an *obj\_ref*, the same algorithm is applied again. In this case, if the second variable is an *obj\_ref*, the first OID value in the first *obj\_set* is stored in the *data* field of the assignment variable. If the second variable is an *obj\_set* then the first OID value which satisfies the above criteria is assigned to the assignment variable.

**loop\_exec():** This function is used for executing the statements inside a for-loop as the number of OID values in an *obj\_set*. The first variable the function reads is referred to as *index variable*, and its type is always an *obj\_ref*. The second variable the function reads is always an *obj\_set* on which the number of iterations for the for-loop are computed. The function opens a new file, *loop\_f* and the statements between the beginning and the end of the for-loop are copied from the *query\_f* to the *loop\_f*. Following this step, a loop is entered. In each iteration of this loop, the data field of the *index variable* is updated with an OID value of the *obj\_set*. Since the number of iterations are equal to the number of OID values in the *obj\_set*, in each iteration a new OID value is stored in the data field of the *index variable*. So, the number of the iteration determines which OID value will be copied from the *obj\_set* to the *index variable*. In each iteration, the function *rtm\_exec()* is called recursively. This time, the *rtm\_exec()* receives the *loop\_f* as its input file.

### C. THE IMPLEMENTATION OF THE PRIMARY OPERATIONS

The RTM performs four types of primary operations, namely, RETRIEVE, DISPLAY, UPDATE and ADD. Once the function *query\_exec()* is called, it reads the next character in the line and branches to the corresponding function which handles a specific primary operation. Figure 5, depicts the flowchart for the *query\_exec()*.

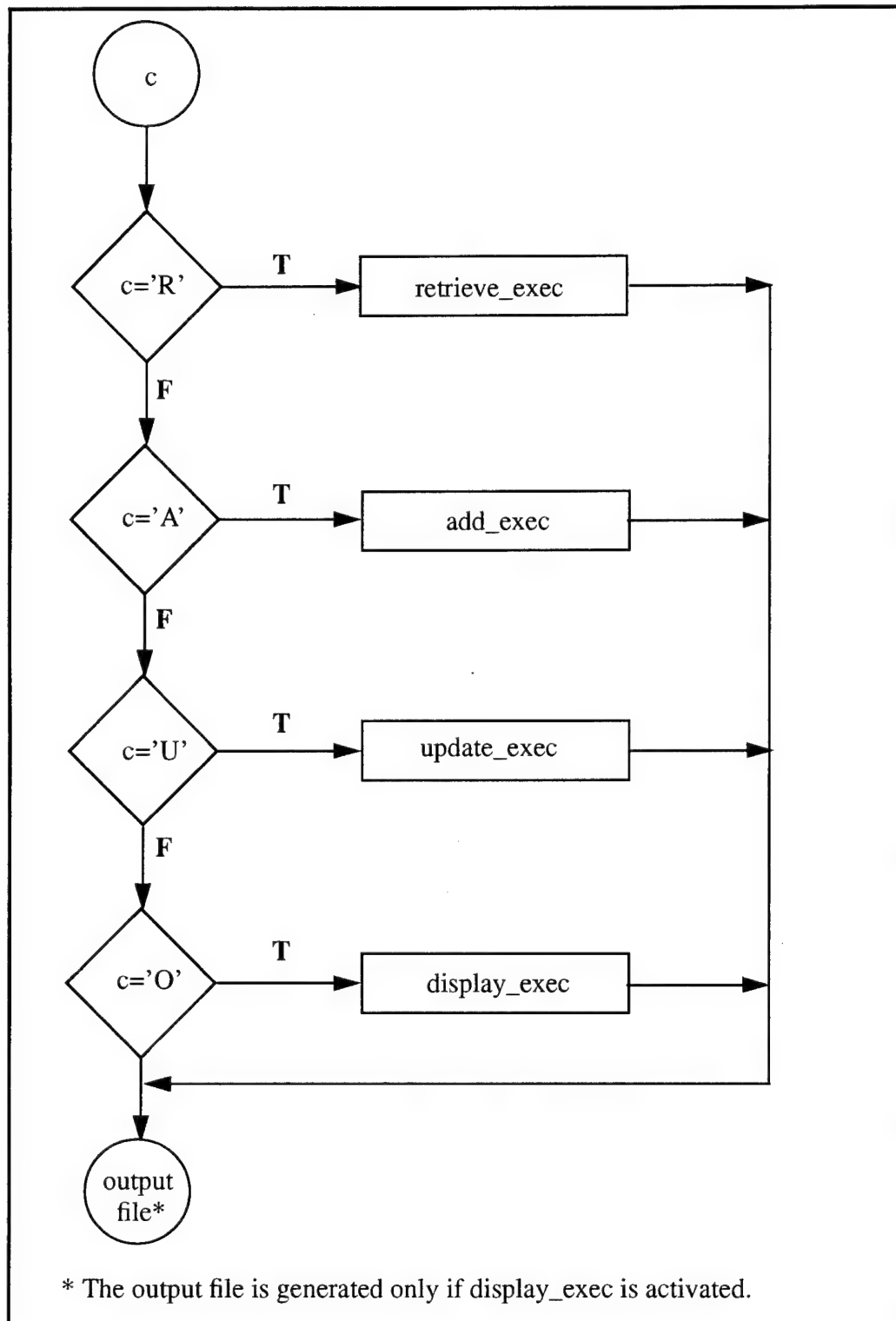


Figure 5: The Flowchart for the query\_exec()



In the rest of this section, each one of these functions will be described.

**retrieve\_exec():** This function is used for retrieving data from the kernel database. First, the function prepares the ABDL query to be sent to the kernel. For this purpose, the ABDL query which is read from the *query\_f* is stored in an array of characters, named *request*. Next, the function checks the status of the flags *flag\_subs\_os* and *flag\_subs\_or* to see if the ABDL query is in well-executable or nearly-executable form. If none of the flags are set, then it means that the query is in well-executable form, and sent to the kernel directly for execution. If the type of the substitution variable is an *obj\_ref*, the ABDL query is modified substituting the variable with its OID value and sent to the kernel. If the type of the substitution variable is an *obj\_set*, then for each OID value in the data field of this *obj\_set*, the ABDL query is modified substituting the variable with the OID value in turn, and sent to the kernel. The responses received from the kernel is stored in the data field of the assignment variable. In case that the type of the assignment variable is an *obj\_ref*, only the first OID value in the file *response\_f* is stored in the assignment variable since its data field is restricted to holding just one OID value. At the end of the function, the substitution and the assignment flags that are found to be set at the beginning, are reset. This function utilizes the ABDL function RETRIEVE. The data retrieved using the function *retrieve\_exec()* is always OID values. For retrieving the actual data to be seen by the user, the function *display\_exec()* is used.

**display\_exec():** This function is used for the retrieval of the data to be displayed to the user. In general, the function follows the same steps taken in the function *retrieve\_exec()*. One difference is that the data retrieved is not assigned to a variable but stored in the file, *output\_f* opened by this function. When the function is activated, it also sets the global flag *flag\_display*. So, when the execution of the *query\_f* is complete, the main program checks the status of this flag and activates the KFS if it is set. Since there may be more than one DISPLAY operation in the *query\_f*, and the KFS is activated after the execution of the whole *query\_f*, all the responses received from the kernel are appended in the *output\_f*. So, the function *display\_exec()* writes an asterisk (\*) in the *output\_f* to

separate the responses to different DISPLAY operations. This function also utilizes the ABDL function RETRIEVE.

**update\_exec():** This function is used to update the data in the kernel database. The steps necessary for the preparation and the transmission of the ABDL query are as in the function retrieve\_exec(). The function makes use of the ABDL function UPDATE.

**add\_exec():** This function is used for building a new relationship between the existing objects in the kernel database. For example, regarding the faculty-student database introduced in [Ref. 8], adding a student to a specific team that he was not a member of before, requires the use of this function. In this situation, that specific student and that specific team already exist in the database but they are not associated. So, the OID value of that student and the OID value of that team are added as a tuple in the table *Student\_team* which keeps track of the students belonging to a team. To distinguish this tuple from the other tuples in the table, the RTM generates an OID value and associates it with this tuple. As stated in [Ref. 9], the tables that implement the *set-of* and the *covering* relationships are generated by the compiler of the object-oriented data definition language and they are hidden from the user. So, the function *add\_exec()* manipulates these kind of tables. Let's assume that the *rtm\_exec()* reads the statement below from the *query\_f*, and calls the function *query\_exec()* which calls the function *add\_exec()* eventually:

[AINSERT(<TEMP,Student\_team>,<OID,?>,<oid\_student,s>,<oid\_team,t>)]

As seen above, the statements using this function are always incomplete ABDL queries and require an OID value and double substitution before being sent to the kernel. The function reads the query until it meets with the *question mark* (?). At this point, the function *get\_objectid()* is called and the OID value generated by this function is replaced with this character. The function *get\_objectid()* gets the time of the day from the computer, parses it into an 8 digit number, and converts this number into a character string. That's why, each OID value is guaranteed to be unique. Then, the function *add\_exec()* continues reading the query and locates the two obj\_refs to be substituted with their OID values. Once

the substitutions are done, the query is ready for execution and sent to the kernel. This function utilizes the ABDL function INSERT.

## V. CONCLUSIONS

The goal of this thesis research is to design and implement a RTM which would complete and execute the ABDL queries, access the database via the kernel system at run time, and prepare the results to be displayed to the user.

I successfully accomplished the task of building a RTM which would perform the functionality described above. The RTM is capable of executing all the object-oriented operations processed by the OODML compiler. The limitations of the RTM and the recommendations for the future research will be detailed in the following sections.

### A. LIMITATIONS

As stated before, the RTM is not a stand-alone software. It is a layer of software linking the object-oriented interface to the kernel system. It is designed to work only for these two software. The structure of the input to the RTM is predefined by way of the pseudocode and the format of the responses coming back from the kernel are known in advance. So, any change in the input to the RTM may hinder its functionality.

The protocol between the RTM and the OODML compiler is very strict and inflexible. The current protocol is sufficient for the object-oriented operations implemented so far. The implementation of the new object-oriented operations may require the generation of new pseudocode.

The RTM does not check the syntactic correctness of an ABDL query received in the *query\_f*. Since the ABDL queries are generated by the OODML compiler itself without user interference, the RTM assumes that they are free of errors.

### B. FUTURE RESEARCH

Some of the object-oriented operations stated in [Ref. 3] are not implemented yet. These operations are *insert*, *delete*, *read-input* and the statistical operations, i.e., *count*, *min*, *max*, *avg*. Regarding the implementation of the statistical operations, there may not be any dependency on the OODML compiler since these operations are simple computations that

may be performed by the RTM itself. On the other side, for the implementation of the other operations, the input that the OODML compiler will provide, is a remarkable factor. For example, the type and the number of the substitutions in the operations *insert* and *delete* are to be determined by the OODML compiler. Unless these issues are fixed, the implementation of these operations may be very fuzzy and time-consuming.

Another feature which has to be added to the RTM in the future is, handling the conditional statements in an object-oriented query. However, before the implementation of this operation in the RTM, the OODML compiler has to be modified to accept this kind of statements and the pseudocode regarding this operation has to be defined.

Although the RTM implemented in this thesis research is built specifically for the object-oriented interface, it may set a good example for the new interfaces to be added to the M<sup>3</sup>DBMS in the future.

## APPENDIX A- THE DEFINITIONS FOR THE PSEUDOCODE

### **% : Declaration of Object Reference**

This pseudocode is used for the declaration of an object reference (obj\_ref). The character string following the '%' specifies the label of the obj\_ref. More than one obj\_refs can be declared on the same line, each being separated by comma. The label of an obj\_ref is limited to three characters. An obj\_ref that is not declared, can not be used in the query\_f.

#### Example:

%ra,rb,i,s,t

### **@ : Declaration of Object Set**

This pseudocode is used for the declaration of an object set (obj\_set). The issues stated above regarding the declaration of an obj\_ref are also valid for the declaration of an obj\_set.

#### Example:

@sa,sb,sc,sd,e

### **& : Assignment Warning**

This pseudocode is used for assignment. It means that the OID values retrieved from the kernel database are to be assigned in the data field of the obj\_ref/obj\_set following the '&'.

#### Example:

&ra

[RETRIEVE((TEMP=Name)and(LNAME=wu)) (OID)]

### **~ : Substitution Warning**

This pseudocode is used for single substitution in an ABDL query. First, the obj\_ref/obj\_set subject to substitution is found in the ABDL query and then, replaced with its OID value(s). There is only one obj\_ref/obj\_set following the '~'.

#### Example:

~sb

[RETRIEVE((TEMP=Course)and(INSTRUCTOR=sb)) (OID)]

### **# : Double Substitution Warning**

This pseudocode is used for double substitution in an ABDL query. There are two arguments following the '#', each separated by comma, and the type of both arguments is obj\_ref. First, the two obj\_refs subject to substitution are located in the ABDL query and then, replaced with their OID values. This kind of substitution occurs only in the ADD operation.

#### Example:

#s,t

[INSERT(<TEMP,Student\_team>,<OID,?>,<oid\_student,s>,<oid\_team,t>)]

### **+ : Union**

This pseudocode is used to compute the disjunction of the OID values stored in two obj\_sets.

#### Example:

&sc

+sa,sb

**\* : Intersection**

This pseudocode is used to compute the conjunction of the OID values stored in two obj\_sets.

Example:

&sd

\*sa,sb

**^ : Get Common**

This pseudocode is used to determine the OID values in the first argument which are repeated for at least the number of OID values in the second argument. The type of the first argument is an obj\_set. The type of the second argument may be either an obj\_set or an obj\_ref.

Example:

&e

^sa,sb

**\$ : For-Loop**

This pseudocode is used for executing the statements inside a for-loop for the number of OID values in the second argument following the '#'. it also marks the beginning of the for-loop. The first argument following the '\$' is called as *index variable* of the for-loop. The type of the first argument is obj\_ref, and the type of the second argument is obj\_set. The example regarding the use of this pseudocode is given following the definition of the '!'.

**! : End of for-loop**

This pseudocode marks the end of the for-loop.

Example:

\$i,e

~i

[ORETRIEVE((TEMP=Course)and(OID=i))(CNAME,CSE\_NO)BY CNAME]

!

## APPENDIX B- THE SAMPLE TRACE OF A QUERY

Script started on Tue Jul 11 12:51:10 1995

//\*\*\*\*\*FACSTUoolreq1\*\*\*\*\*//

Query Display\_Course IS

obj\_set a;

obj\_ref i;

Begin

a := find\_many Course where instructor.pname.lname = 'Wu';

For Each i IN a

display(i.cname, i.cse\_no);

End\_Loop;

End;

pseudocode for real-time monitor

%i

@a,sa,sb,sc

&sa

[RETRIEVE((TEMP=Name)and(LNAME=wu))(OID)]

&sb

~sa

[RETRIEVE((TEMP=Person)and(PNAME=sa))(OID)]

&a

~sb

[RETRIEVE((TEMP=Course)and(INSTRUCTOR=sb))(OID)]

\$i,a

~i

[ORETRIEVE((TEMP=Course)and(OID=i))(CNAME,CSE\_NO)BY CNAME]

!

Successfully parsed!!

real\_time\_monitor() activated

rtm\_exec() activated

create\_obj\_ref() activated

create\_obj\_ref() terminated

create\_obj\_set() activated

create\_obj\_set() terminated

prep\_ass() activated

prep\_ass() terminated

query\_exec() activated

retrieve\_exec() activated

Request= [RETRIEVE((TEMP=Name)and(LNAME=wu))(OID)]

(<OID, N7>)

retrieve\_exec() terminated

query\_exec() terminated

prep\_ass() activated

prep\_ass() terminated

prep\_subs() activated

query\_f



```

prep_subs() terminated
query_exec() activated
retrieve_exec() activated
Request= [RETRIEVE((TEMP=Person)and(PNAME=sa))(OID)]
mod_req =[RETRIEVE((TEMP=Person)and(PNAME=N7))(OID)]
          (<OID, P7>)
retrieve_exec() terminated
query_exec() terminated
prep_ass() activated
prep_ass() terminated
prep_subs() activated
prep_subs() terminated
query_exec() activated
retrieve_exec() activated
Request= [RETRIEVE((TEMP=Course)and(INSTRUCTOR=sb))(OID)]
mod_req =[RETRIEVE((TEMP=Course)and(INSTRUCTOR=P7))(OID)]
          (<OID, C2>)
retrieve_exec() terminated
query_exec() terminated
loop_exec() activated
rtm_exec() activated
prep_subs() activated
prep_subs() terminated
query_exec() activated
display_exec() activated
Request= [RETRIEVE((TEMP=Course)and(OID=i))(CNAME,CSE_NO)BY CNAME]
mod_req =[RETRIEVE((TEMP=Course)and(OID=C2))(CNAME,CSE_NO)BY CNAME]
          (<CNAME, ooprogram, <CSE_NO, 4114>)
display_exec() terminated
query_exec() terminated
rtm_exec() terminated
loop_exec() terminated
rtm_exec() terminated

```

The Result of your Query is :

CNAME	CSE_NO	
-----		
ooprogram	4114	

} The end-result displayed to the user. (The steps taken by the RTM are not displayed, normally.)

```

real_time_monitor() terminated
script done on Tue Jul 11 12:54:56 1995

```

## APPENDIX C- SOURCE CODE FOR THE RTM

```
/* rtm.def.h*/
/* This file includes the structure definitions and the function declarations/

/* Structure holding info about an object reference */
struct obj_ref {
    char name[3];
    char data[20];
};

/* Structure holding info about an object set */
struct obj_set {
    char name[3];
    char data[30][20];
};

/* Structure holding info about the location of an obj_ref/obj_set*/
struct obj_ref_set_info {
    char name[3];
    int index;
};

void real_time_monitor();

void rtm_exec();

void create_obj_ref();

void create_obj_set();

void prep_ass();

void loop_exec();

void prep_subs();

void or_exec();

void and_exec();

void get_common();

void prep_double_subs();

void query_exec();

void retrieve_exec();

void display_exec();

void add_exec();

void update_exec();
```

```

/* rtm.c coded by Erhan SENOC AK (July 95)*/

#include <stdio.h>
#include "rtm_def.h"
#include <licommdata.h>
#include <ool.h>
#include "object_ID.h"

#define FALSE 0
#define TRUE 1

struct obj_ref list_obj_ref[30];
struct obj_set list_obj_set[30];
struct obj_ref_set_info subs_obj_var;
struct obj_ref_set_info list_subs_obj_ref[2];

FILE* fp;
FILE* outfptr;

char c, prev_disp_req[100];

int o_r_ass = FALSE;
int o_s_ass = FALSE;
int flag_display;
int same;
int index ;
int flag_subs_os = FALSE;
int flag_subs_or = FALSE;
int flag_double_subst = FALSE;

void
real_time_monitor(fname)
char* fname;
{
/*  printf("\nreal_time_monitor() activated"); */

    flag_display = FALSE;
    same = FALSE;
    rtm_exec(fname);
/* If the flag is set this means that there is an output to be displayed to
the user. In this case the Kfs is activated, and after the Kfs is called
the output_f is removed */
    if (flag_display) {
        fclose(outfptr);
        system("/u/mdbs/greg/CNTRL/TI/LangIF/src/Obj/Kfs/a.out");
        system("rm /u/mdbs/greg/CNTRL/TI/LangIF/src/Obj/Kfs/output_f");
    }

/*  printf("\nreal_time_monitor() terminated"); */
}

void
rtm_exec(fname)
/* Main function. It opens the query file prepared by the query constructor and
begins to read each line of the file character by character and takes the
appropriate action by comparing the first character of each line with one of
the characters defined below in the switch expression. */

char* fname;
{
    FILE *fopen();

```

```

/*  printf("\nrtm_exec() activated"); */

fp=fopen(fname,"r");
while ((c = getc(fp)) != EOF) {
    switch (c) {
        case '%' : create_obj_ref();
                    break;
        case '@' : create_obj_set();
                    break;
        case '&' : prep_ass();
                    break;
        case '$' : loop_exec();
                    break;
        case '~' : prep_subs();
                    break;
        case '+' : or_exec();
                    break;
        case '*' : and_exec();
                    break;
        case '^' : get_common();
                    break;
        case '#' : prep_double_subs();
                    break;
        case '[' : query_exec();
                    break;
    }
}
fclose(fp);

/*  printf("\nrtm_exec() terminated\n"); */
}

void
create_obj_ref()
/* Reads each obj_ref separated by comma and stores in the
   list_obj_ref array. */
{
    int i,j;

/*  printf("\ncreate_obj_ref() activated"); */

    j=0;
    for (i=0 ; i<30 ; i++) {
        strcpy(list_obj_ref[i].name, "");
        strcpy(list_obj_ref[i].data, "");
    }
    while ((c = getc(fp)) != '\n') {
        i = 0;
        list_obj_ref[j].name[i++] = c;
        while (( c = getc(fp)) != ',') {
            if ( c == '\n')
                break;
            printf("\n%s%c", "c=", c);
            list_obj_ref[j].name[i++] = c;
        }
        list_obj_ref[j].name[i] = '\0';
        j++;
        if ( c == '\n')
            break;
    }
}

```

```

/*  printf("\ncreate_obj_ref() terminated"); */
}

void
create_obj_set()
/* Reads each obj_set separated by comma and stores in the
   list_obj_set array. */
{
    int i,j;

/*  printf("\ncreate_obj_set() activated"); */

    for (i=0 ; i<30 ; i++) {
        strcpy(list_obj_set[i].name, "");
        for (j=0 ; j<30 ; j++)
            strcpy(list_obj_set[i].data[j], "");
    }
    j=0;
    while ((c = getc(fp)) != '\n') {
        i = 0;
        list_obj_set[j].name[i++] = c;
        while ((c = getc(fp)) != ',') {
            if (c == '\n')
                break;
            list_obj_set[j].name[i++] = c;
        }
        list_obj_set[j].name[i] = '\0';
        j++;
        if (c == '\n')
            break;
    }

/*  printf("\ncreate_obj_set() terminated"); */
}

void
prep_ass()
/* This function deals with the intermediate steps necessary for the
   assignment. First, it reads the name of the variable that the assignment will
   be done. Next, the index of this variable in the corresponding list is located
   and stored in the global variable index. */
{
    char ass_var[3];          /* Variable to hold obj_ref/obj_set */
    int i = 0;

/*  printf("\nprep_ass() activated"); */

    while ((c = getc(fp)) != '\n')
        ass_var[i++] = c;
    ass_var[i] = '\0';
    for (i=0 ; i<30 ; i++) {
        if (!(strcmp(list_obj_ref[i].name, ass_var))) {
            index = i;
            o_r_ass = TRUE;
        }
        if (o_r_ass)
            break;
    }
    if (!(o_r_ass)) {
        for (i=0 ; i<30 ; i++) {
            if (!(strcmp(list_obj_set[i].name, ass_var))) {

```

```

        index = i;
        o_s_ass = TRUE;
    }
    if (o_s_ass)
        break;
}

/* printf("\nprep_ass() terminated"); */
}

void
prep_subs()
/* This function handles the intermediate steps necessary for any substitution
   in an ABDL query. First, it reads the name of the variable to be substituted
   with the OID value in its data field. The substitution variable may be either
   an obj_ref or an obj_set. Next, the index of this variable in the corresponding
   list is found and with respect to the type of the variable the flag_subs_or or
   the flag_subs_os is set.*/
{
    int i=0, j;

/* printf("\nprep_subs() activated"); */

    while ((c = getc(fp)) != '\n')
        subs_obj_var.name[i++] = c;
    subs_obj_var.name[i] = '\0';
    for (j=0 ; j<30 ; j++) {
        if (!(strcmp(list_obj_set[j].name, subs_obj_var.name))) {
            subs_obj_var.index = j;
            flag_subs_os = TRUE;
        }
        if (flag_subs_os)
            break;
    }
    for (j=0 ; j<30 ; j++) {
        if (!(strcmp(list_obj_ref[j].name, subs_obj_var.name))) {
            subs_obj_var.index = j;
            flag_subs_or = TRUE;
        }
        if (flag_subs_or)
            break;
    }

/* printf("\nprep_subs() terminated"); */
}

void
prep_double_subs()
/* This function implemented for the need of add_exec() function. Due to the
   characteristic of add operation two substitutions have to be made inside
   the INSERT ABDL request. Beware that this function is not used for any other
   purpose. So, the value of the flag_double_subst is checked only in the
   add_exec() function. */
{
    int i, j=0;

/* printf("\nprep_double_subst() activated"); */

    for ( i=0; i<2 ; i++) {
        strcpy (list_subs_obj_ref[i].name, ""); /* The list_subs_obj_ref is

```

```

        list_subs_obj_ref[i].index = 31;                cleaned from the previous values */
    }
    while ((c = getc(fp)) != '\n') {
        i=0;
        list_subs_obj_ref[j].name[i++] = c;
        while ((c = getc(fp)) != ',') {
            if ( c == '\n')
                break;
            list_subs_obj_ref[j].name[i++] = c;
        }
        list_subs_obj_ref[j].name[i] = '\0';
        j++;
        if ( c == '\n')
            break;
    }
    /* The index of the obj_refs' read above are found searching thru list_obj_ref
       and stored in the list_subs_obj_ref. */
    for (j=0 ; j<2 ; j++) {
        for (i=0; i<30 ; i++) {
            if (!(strcmp(list_obj_ref[i].name , list_subs_obj_ref[j].name))) {
                list_subs_obj_ref[j].index = i;
                break;
            }
        }
    }
    /* Check if the indexes were found and if found assign the flag TRUE */
    if ((list_subs_obj_ref[0].index != 31) && (list_subs_obj_ref[1].index != 31))
        flag_double_subst = TRUE;

    /*  printf("\nprep_double_subst() terminated"); */
}

void
or_exec()
/* This function computes the union of the OID values in two distinct obj_sets
   and stores them in the obj_ref/obj_set referred by the function prep_ass(). */
{
    struct obj_ref_set_info list_or_var[2];
    int i, j=0, n,col=0, k,l, found, not_exist, id, done=FALSE;

    printf("\nor_exec() activated");

    /* Obj_sets to be worked on are put in an array */
    while ((c = getc(fp)) != '\n') {
        i=0;
        list_or_var[j].name[i++] = c;
        while ((c = getc(fp)) != ',') {
            if ( c == '\n')
                break;
            list_or_var[j].name[i++] = c;
        }
        list_or_var[j].name[i] = '\0';
        j++;
        if ( c == '\n')
            break;
    }
    /* The index of the obj_sets read above are found in the list_obj_set */
    for (j=0 ; j<2 ; j++) {
        found = FALSE;
        for (i=0; i<30 ; i++) {
            if (!(strcmp(list_obj_set[i].name , list_or_var[j].name))) {

```

```

        list_or_var[j].index = i;
        found = TRUE;
    }
    if (found)
        break;
}
}
i=0;
id = list_or_var[0].index;
/* If the type of the assignment variable is an obj_set, first, all the OID values
in the first obj_set are assigned to the assignment variable. Then, each OID value
in the second obj_set is compared to the OID values in the assignment variable. If it
does not match with any of them then it is assigned to the assignment variable too. */
if (o_s_ass) {
    while (list_obj_set[id].data[i][col]) {
        strcpy(list_obj_set[index].data[i], list_obj_set[id].data[i]);
        i++;
    }
    j=0;
    k=0;
    while (list_obj_set[list_or_var[1].index].data[j][col]) {
        not_exist = TRUE;
        for (n=0 ; n<i ; n++) {
            if (!(strcmp(list_obj_set[index].data[n],
                        list_obj_set[list_or_var[1].index].data[j])))
                not_exist = FALSE;
        }
        if (not_exist) {
            l = i+k;
            strcpy(list_obj_set[index].data[l], list_obj_set[list_or_var[1].index].data[j]);
            k++;
        }
        j++;
    }
    o_s_ass = FALSE;
    index = 31; /* index can hold a value between 0-29 since the size of the arrays
is 30. Assignment of 31 is a second safety measure else than
the flags o_r_ass and o_s_ass.*/
} /* End of if (o_s_ass) */
/* If the type of the assignment variable is an obj_ref, then the first OID value
of the first obj_set is assigned. If there is no OID value in the first obj_set
then the first OID value in the second obj_set is assigned and the function
terminates */
else if (o_r_ass) {
    if (list_obj_set[id].data[i][col]) {
        strcpy(list_obj_ref[index].data, list_obj_set[id].data[i]);
        done = TRUE;
    }
    if (!(done)) {
        k=0;
        if (list_obj_set[list_or_var[1].index].data[0][col]) {
            strcpy(list_obj_ref[index].data, list_obj_set[list_or_var[1].index].data[0]);
        }
    }
    o_r_ass = FALSE;
    index = 31;
} /* End of if (o_r_ass) */

printf("\nor_exec() terminated");
}

```



```

void
and_exec()
/* This function computes the conjunction of the OID values in two distinct
   obj_sets and stores them in the obj_ref/obj_set referred by the function
   prep_ass() */
{
    struct obj_ref_set_info list_and_var[2];
    int i, j=0, n,col=0, k=0, found;

/*   printf("\nand_exec() activated"); */

    /* Obj_sets to be worked on are put in an array */
    while ((c=getc(fp)) != '\n') {
        i=0;
        list_and_var[j].name[i++] = c;
        while ((c = getc(fp)) != ','){
            if ( c =='\n')
                break;
            list_and_var[j].name[i++] = c;
        }
        list_and_var[j].name[i] = '\0';
        j++;
        if ( c == '\n')
            break;
    }
/* The index of the obj_sets read above are found in the list_obj_set */
    for (j=0 ; j<2 ; j++) {
        found = FALSE;
        for (i=0; i<30 ; i++) {
            if (!(strcmp(list_obj_set[i].name, list_and_var[j].name))) {
                list_and_var[j].index = i;
                found = TRUE;
            }
            if (found)
                break;
        }
    }
/* The data values in each obj_set are compared with eachother and the values
   that match are assigned to the object set specified by the prep_ass() */
    i=0;
    if (o_s_ass) {
        while (list_obj_set[list_and_var[0].index].data[i][col]) {
            j=0;
            found = FALSE;
            while (list_obj_set[list_and_var[1].index].data[j][col]) {
                if (!(strcmp(list_obj_set[list_and_var[0].index].data[i],
                             list_obj_set[list_and_var[1].index].data[j]))) {
                    strcpy(list_obj_set[index].data[k],
                           list_obj_set[list_and_var[0].index].data[i]);
                    k++;
                    found = TRUE;
                }
                if (found)
                    break;
                ++j;
            }
            ++i;
        }
        /* End for while */
    }
    o_s_ass = FALSE;
    index = 31;
}
/* End for if (o_s_ass) */

```

```

if (o_r_ass) {
    while (list_obj_set[list_and_var[0].index].data[i][col]) {
        j=0;
        found = FALSE;
        while (list_obj_set[list_and_var[1].index].data[j][col]) {
            if (!(strcmp(list_obj_set[list_and_var[0].index].data[i],
                           list_obj_set[list_and_var[1].index].data[j]))) {
                strcpy(list_obj_ref[index].data,
                       list_obj_set[list_and_var[0].index].data[i]);
                k++;
                found = TRUE;
            }
            if (found)
                break;
            ++j;
        }
        if (found)
            break; /* Obj_ref assignment */
        ++i;
    } /* End for while */
    o_r_ass = FALSE;
    index = 31;
} /* End for if (o_r_ass) */

/* printf("\nand_exec() terminated"); */
}

void
loop_exec()
/* This function is used for executing the statements inside a for-loop as the
   number of OID values in an obj_set. */
{
    int i = 0, reindex, setindex, j = 0;
    char refname[3], setname[3];
    FILE* fpointer;

    /* printf("\nloop_exec() activated "); */

    /* The first variable read is the index of the for-loop. */
    while ((c = getc(fp)) != ',')
        refname[i++] = c;
    refname[i] = '\0';
    i = 0;
    while ((c = getc(fp)) != '\n')
        setname[i++] = c;
    setname[i] = '\0';

    /* The statements between the beginning and the end of the for-loop are written
       in the file loop_f. */
    fpointer = fopen("loop_f", "w");
    while ((c = getc(fp)) != '!')
        putc(c, fpointer);
    fclose(fpointer);
    c = getc(fp);
    for (i=0 ; i<30 ; i++) {
        if (!(strcmp(list_obj_ref[i].name, refname))) {
            reindex = i;
            break;
        }
    }
    for (i=0; i<30 ; i++) {
        if (!(strcmp(list_obj_set[i].name, setname))) {

```

```

        setindex = i;
        break;
    }
}
i = 0;
fpointer = fp;
/* Loop is entered. The number of iterations are computed with respect to the
number of OID values in the obj_set. In each iteration the rtm_exec() is
called recursively. */
while (list_obj_set[setindex].data[i][j]) {
    strcpy(list_obj_ref[refindex].data, list_obj_set[setindex].data[i]);
    rtm_exec("loop_f");
    i++;
}
fp = fpointer;

/* printf("loop_exec() terminated"); */
}

void
get_common()
/* This function determines the OID values in an obj_set, that are repeated for
at least the number of OID values in another obj_set or in an obj_ref and
assigns these OID values in an obj_ref/obj_set located by the function
prep_ass() */
{
    int i=0, j=0, p=0, k=0, repeat, setsize=0, setindex, arg2size=0, arg2index;
    int done=FALSE;
    char setname[3], arg2name[3];

/* printf("\n get_common() activated "); */

    while ((c = getc(fp)) != ',')
        setname[i++] = c;
    setname[i] = '\0';
    i = 0;
    while ((c = getc(fp)) != '\n')
        arg2name[i++] = c;
    arg2name[i] = '\0';
    for (i=0 ; i<30 ; i++) {
        if (!(strcmp(list_obj_set[i].name, setname))) {
            setindex = i;
            while (list_obj_set[i].data[j++][k])
                setsize = j;
            break;
        }
    }
    j=0;
    for (i=0 ; i<30 ; i++) {
        if (!(strcmp(arg2name, list_obj_set[i].name))) {
            arg2index = i;
            while (list_obj_set[i].data[j++][k])
                arg2size = j;
            done = TRUE;
        }
        else if (!(strcmp(list_obj_ref[i].name, arg2name))) {
            if (list_obj_ref[i].data[k])
                arg2size = 1;
            done = TRUE;
        }
    }
    if (done)

```

```

        break;
    }
done = FALSE;
if (o_s_ass) {
    if (arg2size == 0)
        ;
    else {
        if (arg2size == 1) {
            for ( i=0 ; i<setsize ; i++)
                strcpy(list_obj_set[index].data[i], list_obj_set[setindex].data[i]);
        }
        else {
            for (i=0 ; i<setsize ; i++) {
                repeat = 0;
                for (j=i ; j<setsize ; j++) {
                    if (!(strcmp(list_obj_set[setindex].data[i],
                                list_obj_set[setindex].data[j]))) {
                        repeat++;
                        if (repeat == arg2size)
                            strcpy(list_obj_set[index].data[p++],
                                list_obj_set[setindex].data[i]);
                    }
                }
            }
        }
    }
    o_s_ass = FALSE;
    index = 31;
}
/* End of if (o_s_ass) */
else if (o_r_ass) {
    if (arg2size == 0)
        ;
    else {
        if (arg2size == 1) {
            for ( i=0 ; i<setsize ; i++) {
                strcpy(list_obj_ref[index].data, list_obj_set[setindex].data[i]);
                done = TRUE;
                if (done)
                    break;
            }
        }
        else {
            for (i=0 ; i<setsize ; i++) {
                repeat = 0;
                for (j=i ; j<setsize ; j++) {
                    if (!(strcmp(list_obj_set[setindex].data[i],
                                list_obj_set[setindex].data[j]))) {
                        repeat++;
                        if (repeat == arg2size) {
                            strcpy(list_obj_ref[index].data, list_obj_set[setindex].data[i]);
                            done = TRUE;
                        }
                        if (done)
                            break;
                    }
                }
            }
            if (done)
                break;
        }
    }
}
}
}

```

```

        o_r_ass = FALSE;
        index = 31;
    } /* End of else if (o_r_ass) */
/* printf("\n get_common() terminated "); */
}

void
query_exec()
/* This function is used for branching to a function regarding a specific type of
   data manipulation and retrieval operation. */
{
/* printf("\nquery_exec() activated"); */

    c = getc(fp);
    switch (c) {
    case 'R' : retrieve_exec();
                break;
    case 'O' : display_exec();
                break;
    case 'A' : add_exec();
                break;
    case 'U' : update_exec();
                break;
    }

/* printf("\nquery_exec() terminated"); */
}

void
retrieve_exec()
/* This function retrieves data, specifically, OID values from the kernel database.
   If the type of the substitution variable in the ABDL query stored in the request
   array is an obj_ref, then the ABDL query is modified substituting the variable
   with its OID value and sent to the kernel. If the type of the substitution
   variable is an obj_set then for each OID value in the data field of this obj_set,
   the ABDL query is modified substituting the variable with the OID value in turn
   and sent to the kernel for execution. If there is no substitution in that case
   the whole query is sent directly to the kernel. The responses received back are
   stored in the assignment variable. */
{
    char request[100], req_part1[100], req_part2[100], mod_req[100], temp[30], ch;
    int i=0, j=0, z=0, k, n, l=0, t=0, s, once = 0;
    FILE *fptr, *fopen();

/* printf("\nretrieve_exec() activated"); */

    request[i++] = '[';
    request[i++] = 'R';
    while ((c = getc(fp)) != '\n') {
        if (c == EOF)
            break;
        request[i++] = c;
    }
    request[i] = '\0'; /* ABDL query read in the request */
    i=0;
    if (flag_subs_os) {
        for (k=0 ; k<2 ;k++) {
            while (request[i] != '=')
                req_part1[j++] = request[i++];
            req_part1[j++] = request[i++]; /*Everything till second equality copied*/
        }
    }

```

```

req_part1[j] = '\0';
k = 0;
while (request[i] != '\0')
    temp[k++] = request[i++];
temp[k] = '\0';
while (request[i] != '\0')
    req_part2[z++] = request[i++];
req_part2[z] = request[i];
if (!(strcmp(temp, subs_obj_var.name))) {
    n = 0;
    while (list_obj_set[subs_obj_var.index].data[n][1]) {
        strcpy(mod_req, req_part1);
/* OID value is inserted in the ABDL query */
        strcat(mod_req, list_obj_set[subs_obj_var.index].data[n]);
        strcat(mod_req, req_part2); /*whole request constructed */
        TI_S$TrafUnit(ool_ptr -> oi_curr_db.cdi_dbname, mod_req);
        TI_chk_reqs_left();
        fptr = fopen("/u/mdbs/greg/CNTRL/TI/LangIF/src/Obj/Kc/response_f", "r");
        if (o_s_ass) {
            if (!(once))
                once = t;
            t = 0;
            t = t + once * n;
            while ((ch =getc(fptr)) != EOF){
                while ((ch =getc(fptr)) != ',') {
                    if (ch == EOF)
                        break;
                }
                if (ch == EOF)
                    break;
                ch =getc(fptr); /* space before OID value is skipped */
                s = 0;
                while ((ch =getc(fptr)) != '>') {
                    if (ch == EOF)
                        break;
                    list_obj_set[index].data[t][s++] = ch;
                }
                if (ch == EOF)
                    break;
                list_obj_set[index].data[n][s] = '\0';
                t++;
                if (ch == EOF)
                    break;
            }
        } /* end of if */
    }
else if (o_r_ass) {
    t = 0;
    while ((ch =getc(fptr)) != EOF){
        while ((ch =getc(fptr)) != ',')
            ;
        /* blank space between ',' and OID value skipped */
        ch =getc(fptr);
        while ((ch =getc(fptr)) != '>')
            list_obj_ref[index].data[t++] = ch;
        list_obj_ref[index].data[t] = '\0';
        o_r_ass = FALSE;
        index = 31;
        break; /* while loop broken */
    }
}
}

```

```

        n++;                                /* data[n] incremented */
        fclose(fptr);
    }                                        /* end of while */
    if (o_s_ass) {
        o_s_ass = FALSE;
        index = 31;
    }
}                                        /* End of if (!(strcmp....)) */
flag_subs_os = FALSE;
subs_obj_var.index = 31;
}                                        /* End of if(flag_subs_os) */
else if (flag_subs_or) {
    for ( k=0 ; k<2 ;k++) {
        while (request[i] != '=')
            req_part1[j++] = request[i++];
        req_part1[j++] = request[i++]; /*Everything till second equality copied*/
    }
    req_part1[j] = '\0';
    k = 0;
    while (request[i] != '\0')
        temp[k++] = request[i++];
    temp[k] = '\0';
    while (request[i] != '\0')
        req_part2[z++] = request[i++];
    req_part2[z] = request[i];
    if (!(strcmp(temp, subs_obj_var.name))) {
        if ( list_obj_ref[subs_obj_var.index].data[1]) {
            strcpy( mod_req, req_part1);
            strcat( mod_req, list_obj_ref[subs_obj_var.index].data);
            strcat( mod_req, req_part2); /* whole request constructed */
            TI_SSTrafUnit( ool_ptr -> oi_curr_db.cdi_dbname, mod_req);
            TI_chk_reqs_left();
            fptr = fopen("/u/mdbs/greg/CNTRL/TI/LangIF/src/Obj/Kc/response_f","r");
            if (o_s_ass) {
                t=0;
                while (( ch = getc(fptr)) != EOF){
                    while (( ch = getc(fptr)) != ',') {
                        if (ch == EOF)
                            break;
                    }
                    if (ch == EOF)
                        break;
                    ch = getc(fptr); /* space before OID value is skipped */
                    s = 0;
                    while (( ch = getc(fptr)) != '>') {
                        if ( ch == EOF)
                            break;
                        list_obj_set[index].data[t][s++] = ch;
                    }
                    if (ch == EOF)
                        break;
                    list_obj_set[index].data[t][s] = '\0';
                    t++;
                    if ( ch == EOF)
                        break;
                }
                o_s_ass = FALSE;
                index = 31;
            }
        }
    }
}
else if (o_r_ass) {

```

```

        t = 0;
        while (( ch = getc(fp_ptr)) != EOF){
            while (( ch = getc(fp_ptr)) != ',')
                ;
            /* blank space between ',' and OID value skipped */
            ch = getc(fp_ptr);
            while (( ch = getc(fp_ptr)) != '>')
                list_obj_ref[index].data[t++] = ch;
            list_obj_ref[index].data[t] = '\0';
            o_r_ass = FALSE;
            index = 31;
            break;
        }
        /* while loop broken */
    }
    fclose(fp_ptr);
}
/* end of if */
/* End of if (!(strcmp....)) */
flag_subs_or = FALSE;
subs_obj_var.index = 31;
}
/* End of if(flag_subs_or) */
else {
    TI_SSTrafUnit( ool_ptr -> oi_curr_db.cdi_dbname, request);
    TI_chk_reqs_left();
    fp_ptr = fopen("/u/mdbs/greg/CNTRL/TI/LangIF/src/Obj/Kc/response_f" , "r");
    if (o_s_ass) {
        t = 0;
        while (( ch = getc(fp_ptr)) != EOF){
            while (( ch = getc(fp_ptr)) != ',') {
                if ( ch == EOF )
                    break;
            }
            ;
        }
        if ( ch == EOF )
            break;
        ch = getc(fp_ptr);
        /* space before OID value is skipped */
        s = 0;
        while (( ch = getc(fp_ptr)) != '>') {
            if ( ch == EOF )
                break;
            list_obj_set[index].data[t][s++] = ch;
        }
        if ( ch == EOF )
            break;
        list_obj_set[index].data[t][s] = '\0';
        t++;
    }
    o_s_ass = FALSE;
    index = 31;
}
else if (o_r_ass) {
    t = 0;
    while (( ch = getc(fp_ptr)) != EOF){
        while (( ch = getc(fp_ptr)) != ',')
            ;
        ch = getc(fp_ptr); /* blank space between ',' and OID value skipped */
        while (( ch = getc(fp_ptr)) != '>')
            list_obj_ref[index].data[t++] = ch;
        list_obj_ref[index].data[t] = '\0';
        o_r_ass = FALSE;
        index = 31;
        break;
    }
    /* while loop broken */
}

```



```

    }
}
fclose(fptr);
}

/* printf("\nretrieve_exec() terminated"); */
}

void
display_exec()
/* This function retrieves the data to be displayed to the user. */
{
    char ch,request[100], req_part1[100],req_part2[100], mod_req[100], temp[30];
    int i=0, j=0, z=0, k, n, l=0;
    FILE* fopen();
    FILE* fptr;

/* printf("\ndisplay_exec() activated"); */

    request[i++] = '[';
    while ((c =getc(fp)) != '\n') {
        if ( c == EOF)
            break;
        request[i++] = c;
    }
    request[i] = '\0'; /* ABDL query read in the request */
    if (!(flag_display)) {
/* The responses are stored in the output_f */
        outfptr = fopen("/u/mdbs/greg/CNTRL/TI/LangIF/src/Obj/Kfs/output_f", "a");
        flag_display = TRUE;
        strcpy(prev_disp_req, request);
    }
/* The flag same is used to separate the responses to different DISPLAY operations
in the same query_f. */
    if (!(strcmp(prev_disp_req, request)))
        same = TRUE;
    else {
        strcpy(prev_disp_req, request);
        same = FALSE;
    }
    i=0;
    if (flag_subs_os) {
        for ( k=0 ; k<2 ;k++) {
            while (request[i] != '=')
                req_part1[j++] = request[i++];
            req_part1[j++] = request[i++]; /*Everything till second equality copied*/
        }
        req_part1[j] = '\0';
        k = 0;
        while (request[i] != ')')
            temp[k++] = request[i++];
        temp[k] = '\0';
        while (request[i] != '\0')
            req_part2[z++] = request[i++];
        req_part2[z] = request[i];
        if (!(strcmp(temp, subs_obj_var.name))) {
            n =0;
            while ( list_obj_set[subs_obj_var.index].data[n][1]) {
                strcpy( mod_req, req_part1);
                strcat( mod_req, list_obj_set[subs_obj_var.index].data[n]);
                strcat( mod_req, req_part2); /* whole request constucted */
            }
        }
    }
}

```

```

        TI_S$TrafUnit( ool_ptr -> oi_curr_db.cdi_dbname, mod_req);
        TI_chk_reqs_left();
        if (!(same))
            putc('*', outfptr);
        fptr = fopen("/u/mdbs/greg/CNTRL/TI/LangIF/src/Obj/Kc/response_f", "r");
        while ((ch = getc(fptr)) != EOF)
            putc(ch, outfptr);
        fclose(fptr);
        n++;
    }
    /* data[n] incremented */
    /* end of while */
    /* End of if (!(strcmp....)) */
}
flag_subs_os = FALSE;
subs_obj_var.index = 31;
/* End of if(flag_subs_os) */
else if (flag_subs_or) {
    for (k=0 ; k<2 ;k++) {
        while (request[i] != '=')
            req_part1[j++] = request[i++];
        req_part1[j++] = request[i++]; /*Everything till second equality copied*/
    }
    req_part1[j] = '\0';
    k = 0;
    while (request[i] != ')')
        temp[k++] = request[i++];
    temp[k] = '\0';
    while (request[i] != '\0')
        req_part2[z++] = request[i++];
    req_part2[z] = request[i];
    if (!(strcmp(temp, subs_obj_var.name))) {
        if (list_obj_ref[subs_obj_var.index].data[1]) {
            strcpy(mod_req, req_part1);
            strcat(mod_req, list_obj_ref[subs_obj_var.index].data);
            strcat(mod_req, req_part2); /* whole request constructed */
            TI_S$TrafUnit( ool_ptr -> oi_curr_db.cdi_dbname, mod_req);
            TI_chk_reqs_left();
            if (!(same))
                putc('*', outfptr);
            fptr = fopen("/u/mdbs/greg/CNTRL/TI/LangIF/src/Obj/Kc/response_f", "r");
            while ((ch = getc(fptr)) != EOF)
                putc(ch, outfptr);
            fclose(fptr);
        }
        /* end of if */
    }
    /* End of if (!(strcmp....)) */
    flag_subs_or = FALSE;
    subs_obj_var.index = 31;
    /* End of if(flag_subs_or) */
}
else {
    TI_S$TrafUnit( ool_ptr -> oi_curr_db.cdi_dbname, request);
    TI_chk_reqs_left();
    if (!(same))
        putc('*', outfptr);
    fptr = fopen("/u/mdbs/greg/CNTRL/TI/LangIF/src/Obj/Kc/response_f", "r");
    while ((ch = getc(fptr)) != EOF)
        putc(ch, outfptr);
    fclose(fptr);
}
/* end for else */

/* printf("\ndisplay_exec() terminated"); */
}

```

```

void
update_exec()
/* For updating the data in the kernel database. Similar steps as in the function
retrieve_exec() are taken. */
{
    char request[100], req_part1[100], req_part2[100], mod_req[100], temp[30];
    int i=0, j=0, z=0, k, n, l=0;

/*   printf("\nupdate_exec() activated"); */

    request[i++] = '[';
    request[i++] = 'U';
    while ((c = getc(fp)) != '\n') {
        if (c == EOF)
            break;
        request[i++] = c;
    }
    request[i] = '\0';    /* ABDL query read in the request */
    i=0;
    if (flag_subs_os) {
        for (k=0; k<2; k++) {
            while (request[i] != '=')
                req_part1[j++] = request[i++];
            req_part1[j++] = request[i++]; /*Everything till second equality. copied */
        }
        req_part1[j] = '\0';
        k = 0;
        while (request[i] != ')')
            temp[k++] = request[i++];
        temp[k] = '\0';
        while (request[i] != '\0')
            req_part2[z++] = request[i++];
        req_part2[z] = request[i];
        if (!(strcmp(temp, subs_obj_var.name))) {
            n = 0;
            while (list_obj_set[subs_obj_var.index].data[n][1]) {
                strcpy(mod_req, req_part1);
                strcat(mod_req, list_obj_set[subs_obj_var.index].data[n]);
                strcat(mod_req, req_part2);    /* whole request constructed */
                TI_SS$TrafUnit(ool_ptr -> oi_curr_db.cdi_dbname, mod_req);
                TI_chk_reqs_left();
                n++;
            }
            /* data[n] incremented */
            /* end of while */
            /* End of if (!(strcmp....)) */
        }
        flag_subs_os = FALSE;
        subs_obj_var.index = 31;
    }
    /* End of if(flag_subs_os) */

    else if (flag_subs_or) {
        for (k=0; k<2; k++) {
            while (request[i] != '=')
                req_part1[j++] = request[i++];
            req_part1[j++] = request[i++]; /*Everything till second equality copied */
        }
        req_part1[j] = '\0';
        k = 0;
        while (request[i] != ')')
            temp[k++] = request[i++];
        temp[k] = '\0';
        while (request[i] != '\0')
            req_part2[z++] = request[i++];
        req_part2[z] = request[i];
    }
}

```

```

        if (!(strcmp(temp, subs_obj_var.name))) {
            if (list_obj_ref[subs_obj_var.index].data[1]) {
                strcpy(mod_req, req_part1);
                strcat(mod_req, list_obj_ref[subs_obj_var.index].data);
                strcat(mod_req, req_part2); /* whole request constructed */
                TI_SSTrafUnit(ool_ptr -> oi_curr_db.cdi_dbname, mod_req);
                TI_chk_reqs_left();
            } /* end of if */
        } /* End of if (!(strcmp....)) */
        flag_subs_or = FALSE;
        subs_obj_var.index = 31;
    } /* End of if(flag_subs_or) */
else {
    TI_SSTrafUnit(ool_ptr -> oi_curr_db.cdi_dbname, request);
    TI_chk_reqs_left();
} /* End for else */

/* printf("\nupdate_exec() terminated"); */
}

void
add_exec()
/* Used for building a new relationship between the existing objects in the kernel
   database, not inserting a new object in the database. Before the query is
   sent to the kernel for execution, it is modified by double substitution and the
   addition of an OID value.*/
{
    char request[100], req_part1[100], req_part2[100], mod_req[100], temp[30];
    int i=0, j=0, k, z=0, l=0;

    /* printf("\nadd_exec() activated"); */

    request[i++] = '[';
    while ((c = getc(fp)) != '\n') {
        if (c == EOF)
            break;
        request[i++] = c;
    }
    request[i] = '\0'; /* ABDL query read in the request */
    if (flag_double_subst) {
        i=0;
        while (request[i] != '?')
            req_part1[j++] = request[i++];
        req_part1[j] = '\0';
        j=0;
        i++;
        while (request[i] != '\0')
            req_part2[j++] = request[i++];
        req_part2[j] = '\0';
        strcpy(mod_req, req_part1);
        /* OID value is added in the place of '?' in the query. */
        strcat(mod_req, get_objectid());
        strcat(mod_req, req_part2);
        i=j=0;
        /* The first substitution is handled below */
        for (k=0; k<5; k++) {
            while (mod_req[i] != ',')
                req_part1[j++] = mod_req[i++];
            req_part1[j++] = mod_req[i++];
        }
        req_part1[j] = '\0';
    }

```

```

k=0;
while (mod_req[i] != '>')
    temp[k++] = mod_req[i++];
temp[k] = '\0';
while (mod_req[i] != '\0')
    req_part2[z++] = mod_req[i++];
req_part2[z] = '\0';
for ( i=0 ; i<2 ; i++) {
    if (!(strcmp(temp, list_subs_obj_ref[i].name))) {
        if (list_obj_ref[list_subs_obj_ref[i].index].data[1]) {
            strcpy(mod_req, req_part1);
            strcat(mod_req, list_obj_ref[list_subs_obj_ref[i].index].data);
            strcat(mod_req, req_part2);
        }
        break;
    }
}
/* The second substitution is handled below */
i=j=z=0;
for (k=0 ; k<7 ; k++) {
    while (mod_req[i] != ',')
        req_part1[j++] = mod_req[i++];
    req_part1[j] = '\0';
    k=0;
    while (mod_req[i] != '>')
        temp[k++] = mod_req[i++];
    temp[k] = '\0';
    while (mod_req[i] != '\0')
        req_part2[z++] = mod_req[i++];
    req_part2[z] = '\0';
    for ( i=0 ; i<2 ; i++) {
        if (!(strcmp(temp, list_subs_obj_ref[i].name))) {
            if (list_obj_ref[list_subs_obj_ref[i].index].data[1]) {
                strcpy(mod_req, req_part1);
                strcat(mod_req, list_obj_ref[list_subs_obj_ref[i].index].data);
                strcat(mod_req, req_part2);
            }
            break;
        }
    }
}
TI_SSTrafUnit( ool_ptr -> oi_curr_db.cdi_dbname, mod_req);
TI_chk_reqs_left();
flag_double_subst = FALSE;
} /* End of if */

/* printf("\nadd_exec() terminated"); */
}

```

## LIST OF REFERENCES

1. Hsiao, David K., "Interoperating and Integrating the Multidatabase and Systems," presented at ACM CSC'95, Nashville, Tennessee, Mar. 1995.
2. J.V. Joseph et al., "Object-Oriented Databases: Design and Implementation," IEEE Proceedings, vol. 79, pp. 42-63, Jan. 1991.
3. Stephens, M.W., *Design and Specification of an Object-Oriented Data Manipulation Language*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1995.
4. Kellett, D. and Kwon, T., *The Instrumentation of a Kernel DBMS for the Support of a Database in the O-ODDL Specification*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1995.
5. Clark, R. and Yildirim, N., *The Instrumentation of a Kernel DBMS for the Execution of Kernel Transactions Equivalent to their O-O Transactions*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1995.
6. Elmasri and Navathe, *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company, Inc., 1990.
7. Barbosa, C. and Kutlusan, A., *The Design and Implementation of a Compiler for the Object-Oriented Data Manipulation Language (O-ODML Compiler)*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1995.
8. Badgett, R.B., *The Design and Specification of an Object-Oriented Data Definition Language (O-ODDL)*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1995.
9. Ramirez, L. and Tan, R.M., *The Design and Implementation of a Compiler for the Object-Oriented Data Definition Language (O-ODDL Compiler)*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1995.



## INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center ..... 2  
     Cameron Station  
     Alexandria, VA 22304-6145
  
2. Dudley Knox Library ..... 2  
     Code 013  
     Naval Postgraduate School  
     Monterey, CA 93943-5101
  
3. Chairman, Code CS ..... 2  
     Computer Science Department  
     Naval Postgraduate School  
     Monterey, CA 93943
  
4. Dr David K. Hsiao, Code CS/HS ..... 2  
     Computer Science Department  
     Naval Postgraduate School  
     Monterey, CA 93943
  
5. Dr C. Thomas Wu, Code CS/KA ..... 2  
     Computer Science Department  
     Naval Postgraduate School  
     Monterey, CA 93943
  
6. Ms. Doris Mlezko ..... 2  
     Code P22305  
     Weapons Division  
     Naval Air Warfare Center  
     Pt Mugu, CA 93042-5001
  
7. Ms. Sharon Cain ..... 2  
     NAIC/SCDD  
     4115 Hebble Creek Rd  
     Wright Patterson AFB, OH 45433-5622
  
8. Deniz Kuvvetleri Komutanligi ..... 2  
     Bakanliklar-Ankara 06600  
     Turkey



9. Ltjg Erhan Senocak ..... 2  
1671/4 sok. No:11/8  
Karsiyaka-Izmir 35601  
Turkey